Regular Expressions and Automata

Berlin Chen 2003

References:
1. Speech and Language Processing, chapter 2
Introduction

- Regular Expressions (REs)
- Finite-State Automata (FSAs)
- Formal Languages
- Deterministic vs. Nondeterministic FSAs
- Concatenation and union of FSAs
- Finite-State Transducers (FSTs)
- FSTs for Morphology Parsing
- Probabilistic FSTs
Regular Expressions (REs)

• First developed by Kleene in 1956
• Definition
  – A formula in a special (meta-) language that is used for specifying simple classes of strings
    • A string is any sequence of alphanumERIC characters (letters, numbers, spaces, tabs, and punctuation)
      • Are case sensitive
  – An algebraic notation for characterizing a set of strings
    • Specify search strings in Web IR systems
    • Define a language in a formal way
Basic Regular Expression Patterns

- Regular expression search requires a pattern that we want to search for, and a corpus of texts to search through
  - Search through the corpus returning all texts (all matches or only the first match) contain the pattern (returning the line of document)

<table>
<thead>
<tr>
<th>RE</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/woodchucks/</td>
<td>“interesting links to woodchucks and lemurs”</td>
</tr>
<tr>
<td>/a/</td>
<td>“Mary Ann stopped by Mona’s”</td>
</tr>
<tr>
<td>/Chaire_says,/</td>
<td>“Dagmar, my gift please, Chaire says, ”</td>
</tr>
<tr>
<td>/song/</td>
<td>“All our pretty songs”</td>
</tr>
<tr>
<td>/!/</td>
<td>“You’ve left the burglar behind again !” said Nori</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

• **Square braces [ and ]**
  - The string of characters inside the **braces** specify a disjunction of characters

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ woodchuck/</td>
<td>Woodchuck or woodchuck</td>
<td>“Woodchuck”</td>
</tr>
<tr>
<td>/ [abc]/</td>
<td>‘a’, ‘b’, or ‘c’</td>
<td>“In uomini, in soldati”</td>
</tr>
<tr>
<td>/ [1234567890]/</td>
<td>any digit</td>
<td>“plenty of 7 to 5”</td>
</tr>
</tbody>
</table>

• **Dash (-) specifies any one character in a range**

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[A-Z]/</td>
<td>an uppercase letter</td>
<td>“we should call it “Drenched Blossoms””</td>
</tr>
<tr>
<td>/[a-z]/</td>
<td>a lowercase letter</td>
<td>“my beans were impatient to be hoed!”</td>
</tr>
<tr>
<td>/[0-9]/</td>
<td>a single digit</td>
<td>“Chapter 1: Down the Rabbit Hole”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

- **Caret (^)** specifies what a single character cannot be in the square braces

<table>
<thead>
<tr>
<th>RE</th>
<th>Match (single characters)</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>[^A-Z]</td>
<td>not an uppercase letter</td>
<td>“Oyfn pripechik”</td>
</tr>
<tr>
<td>[^Ss]</td>
<td>neither ‘S’ nor ‘s’</td>
<td>“I have no exquisite reason for’t”</td>
</tr>
<tr>
<td>[^.]</td>
<td>not a period</td>
<td>“our resident Djinn”</td>
</tr>
<tr>
<td>[e^]</td>
<td>either ‘e’ or ‘^’</td>
<td>“look up ^ now”</td>
</tr>
<tr>
<td>a^b</td>
<td>the pattern ‘a^b’</td>
<td>“look up a^b now”</td>
</tr>
</tbody>
</table>

- **Question-mark (?)** specify zero or one instances of the previous character

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>woodchucks?</td>
<td>woodchuck or woodchucks</td>
<td>“woodchuck”</td>
</tr>
<tr>
<td>colou?r</td>
<td>color or colour</td>
<td>“colour”</td>
</tr>
</tbody>
</table>
Basic Regular Expression Patterns

• **Kleene star (**) means zero or more occurrences of the immediately previous character or regular expression**
  – E.g.: the sheep language
    
    `/baaa*/`
  – Multiple digits
    
    `/[0-9][0-9]*/`

• **Kleene + (+) means one or more occurrences of the immediately previous character or regular expression**
  – E.g.: the sheep language
    
    `/baa+!/`
  – Multiple digits
    
    `/[0-9]+/`
Basic Regular Expression Patterns

• Period (.) is used as a wildcard expression that matches any single character (except a carriage return)

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>/beg\s.n/</td>
<td>any character between \beg and \n</td>
<td>begin, \beg\n, begun</td>
</tr>
</tbody>
</table>

– Often used together with Kleene star (*) to specify any string of characters
  • E.g.: find line in which a particular word appears twice
    /aardvark.* aardvark/
Basic Regular Expression Patterns

• Anchors are special characters that anchor regular expressions to particular places in a string
  – The caret (^) also can be used to match the start of a line
  – Three usages of the caret: to match the start of a line, negation inside of square braces, and just to mean caret
  – The dollar sign ($) match the end of a line
  – (\b) matches a word boundary while (\B) matches a non-boundary

– E.g. :/^The dog\.$/ matches a line contains only the phrase The dog.
Disjunction

• The pipe symbol (|) specifies the disjunction operation
  – E.g.: match either cat or dog
    /cat|dog/
  – Specify singular and plural nouns
    /gupp(y|ies)/
Precedence

• Operator precedence hierarchy

<table>
<thead>
<tr>
<th>Precedence</th>
<th>( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counters</td>
<td>* + ? {}</td>
</tr>
<tr>
<td>Sequences and anchors</td>
<td>the ^my end$</td>
</tr>
<tr>
<td>Disjunction</td>
<td></td>
</tr>
</tbody>
</table>
A More Complex Example

• Example: Deal with prices, $199, $199.99, etc., with decimal point and two digits afterwards

```
\b\$\[0-9]+(\.[0-9][0-9])?\b/
```

Don't mean end-of-line here.  match a word boundary

• Example: Deal with processor speed (in MHz or GHz), disk space (in Gb), or memory size (in Mb or Gb)

```
\b\[0-9]+\*\((MHz[Mm]egahertz|GHz|Gg]igahertz\)\b/
```

```
\b\[0-9]+\*\((Mb[Mm]egabytes\?|Gb|Gg]egabytes\?)\b/
```
Advanced Operators

• Useful aliases for common ranges

<table>
<thead>
<tr>
<th>RE</th>
<th>Expansion</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0-9]</td>
<td>any digit</td>
<td>Party_of_5</td>
</tr>
<tr>
<td>\D</td>
<td>[^0-9]</td>
<td>any non-digit</td>
<td>Blue_moon</td>
</tr>
<tr>
<td>\w</td>
<td>[a-zA-Z0-9_]</td>
<td>any alphanumeric or space</td>
<td>Daiyu</td>
</tr>
<tr>
<td>\W</td>
<td>[^\w]</td>
<td>a non-alphanumeric</td>
<td>!!!</td>
</tr>
<tr>
<td>\s</td>
<td>[\r\t\n\f]</td>
<td>whitespace (space, tab)</td>
<td>in_Concord</td>
</tr>
<tr>
<td>\S</td>
<td>[^\s]</td>
<td>Non-whitespace</td>
<td></td>
</tr>
</tbody>
</table>

• Regular expression for counting

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>zero or more occurrences of the previous char or expression</td>
</tr>
<tr>
<td>+</td>
<td>one or more occurrences of the previous char or expression</td>
</tr>
<tr>
<td>?</td>
<td>exactly zero or one occurrence of the previous char or expression</td>
</tr>
<tr>
<td>{n}</td>
<td>n occurrences of the previous char or expression</td>
</tr>
<tr>
<td>{n,m}</td>
<td>from n to m occurrences of the previous char or expression</td>
</tr>
<tr>
<td>{n,}</td>
<td>at least n occurrences of the previous char or expression</td>
</tr>
</tbody>
</table>
Characters need to be backslashed

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>an asterisk “*”</td>
<td>“K<em>A</em>P<em>L</em>A*N”</td>
</tr>
<tr>
<td>.</td>
<td>a period “.”</td>
<td>“Dr. Livingston, I presume”</td>
</tr>
<tr>
<td>?</td>
<td>a question mark</td>
<td>“Would you light my candle?”</td>
</tr>
<tr>
<td>\n</td>
<td>a newline</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>a tab</td>
<td></td>
</tr>
</tbody>
</table>
Substitution and Memory

- Substitution operator \texttt{s/regexp1/regexp2/} allow a string characterized by one regular expression to be replaced by a string characterized by a different one
  \begin{itemize}
  \item \texttt{s/colour/color/}
  \item Refer to a particular subpart of the string matching the first pattern, e.g., put angle brackets around all integers in a text (using parenthesis and number operators)
    \begin{itemize}
    \item \texttt{s/([0-9]+)/<\1>/}
    \item Specify a certain string or expression occurs twice in the text (the \texttt{Xer} they \texttt{were}, the \texttt{Xer} they \texttt{will be})
      \begin{itemize}
      \item \texttt{/the (.*) er they \texttt{were}, the \texttt{\1er} they \texttt{will be/}
      \item \texttt{/the (.*) er they (.*), the \texttt{\1er they \2/}
    \end{itemize}
  \end{itemize}
  \end{itemize}
Substitution and Memory

• *Substitution using memory* is not part of every regular expression language and is often considered as “extended” feature of regular expressions

• *Substitution using memory* is very useful in implementing simple natural understanding systems
Example: ELIZA

• A simple natural-language understanding program

<table>
<thead>
<tr>
<th>User1: Men are all alike.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELIZA1: IN WHAT WAY</td>
</tr>
<tr>
<td>User2: They’re always bugging us about something or other.</td>
</tr>
<tr>
<td>ELIZA2: CAN YOU THINK OF A SPECIAL EXAMPLE</td>
</tr>
<tr>
<td>User3: Well, my boyfriend made me come here.</td>
</tr>
<tr>
<td>ELIZA3: YOUR BOYFRIEND MADE YOU COME HERE</td>
</tr>
<tr>
<td>User4: He says I’m depressed much of the time.</td>
</tr>
<tr>
<td>ELIZA4: I AM SORRY TO HERE YOU ARE DEPRESSED</td>
</tr>
</tbody>
</table>

s/.* I’m (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/ 

s/.* all .*/IN WHAT WAY/ 

s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
Finite-State Automata (FSAs)

- FSA are the theoretical foundation of a good deal of the computational work
  - A directed graph with a finite set of vertices (nodes) as well as arcs (links) between pairs of vertices
  - An FSA can be used for recognizing (accepting) a set of strings (the input written on a long tape)
  - An FSA can be represented with a state-transition table

An FSA.

A tape with cells.

The state-transition table
Finite-State Automata (FSAs)

• FSAs and REs
  – Any RE can be implemented as a FSA (except REs with memory feature)
  – Any FSA can be described with a RE (REs can be viewed as a textual way of specifying the structure of FSAs)
  – Both REs and FSAs can be used to describe regular languages

• The main theme in the course
  – Introduce the FSAs for some REs
  – Show how the mapping from REs to FSAs proceeds
Sheep FSA

• We can say the following things about this machine, /baa+!/
  - It has 5 states
  - At least b, a, and ! are in its alphabet
  - $q_0$ is the start state
  - $q_4$ is an accept state
  - It has 5 transitions

---

```
baa!
baaa!
baaaaa!
baaaaaa!
baaaaaaa!
....
```
Formal Definition of FSAs

• We can specify an FSA by enumerating the following 5 things
  - \( Q \): the set of states, \( Q = \{ q_0, q_1, \ldots, q_N \} \)
  - \( \Sigma \): a finite alphabet of symbols
  - \( q_0 \): a start/initial state
  - \( F \): a set of accept/final states
  - \( \delta (q,i) \): a transition function that maps \( Q \times \Sigma \) to \( Q \)

• Deterministic (FSAs/Recognizers)
  - Has no choice points, the automata/algorithms always know what to do for any input
  - The behavior during recognition is fully determined by the state it is in and the symbol it is looking at
Formal Definition of FSAs

• **What is “recognition”**
  - The process of determining if a string should be accepted by a machine
  - Or, it is the process of determining if a string is in the language defined with the machine
  - Or, it is the process of determining if a regular expression matches a string

• **The recognition process**
  - Simply a process of starting in the start state
  - Examine the current input
  - Consult the table
  - Go to a new state and updating the tape pointer
  - Continue until you run out of tape
Algorithm for Deterministic FSAs

function D-RECOGNIZE(tape, machine) returns accept or reject

    index ← Beginning of tape
    current-state ← Initial state of machine
    loop
        if End of input has been reached then
            if current-state is an accept state then
                return accept
            else
                return reject
        elseif transition-table[current-state, tape[index]] is empty then
            return reject
        else
            current-state ← transition-table[current-state, tape[index]]
            index ← index + 1
        end
Adding a Fail State to the FSA

The fail/sink state.
Formal Languages

- Sets of strings composed of symbols from a finite-set (alphabet) and permitted by the rules of formation
- A model (e.g. FSA) which can both generate and recognize (accept) all and only the strings of a formal language
  - A definition of the formation language (without having to enumerating all strings in the language)
  - Given a model $m$, we can use $L(m)$ to mean “the formal language characterized by $m$”
  - The formal language defined by the sheeptalk FSA $m$ $L(m) = \{bbaa!, baa!, baaaa!, baaaaaa!, \ldots\}$
- Often use formal languages to model phonology, morphology, or syntax, …
FSA Dealing with Dollars and Cents

- Such a formal language would model the subset of English.

Account for number from 1 to 99.

Account for number from 1 to 99.
Two Perspectives for FSAs

• FSAs are acceptors that can tell you if a string is in the language
  - Parsing: find the structure in the string
• FSAs are generators to produce all and only the strings in the language
  - Production/generation: produce a surface form
Non-Deterministic FSAs

• Non-Deterministic FSAs: NFSAs
• Recall
  – “Deterministic” means the behavior during recognition is fully determined by the state it is in and the symbol it is looking at

• E.g.: non-deterministic FSAs for the sheeptalk
Non-Deterministic FSAs

• With $\varepsilon$ transitions
  – Arcs that have no symbols on them
    • Move without looking at the input

• When NFSAs take a wrong choice
  – Follow the wrong arc and reject the input when we should have accepted it
    • E.g. when input is “bad!”
Solutions for Wrong Choices

• **Backup**
  – When at a choice point, put a marker (current state, current position at the input tape) and unexplored choices on the agenda
  • Remember all alternatives

• **Look-ahead**
  – We could look ahead in the input to help us decide which path to take

• **Parallelism**
  – When at a choice point, we could look at every alternative path in parallel
Algorithm for Non-Deterministic FSAs

function ND-RECOGNIZE(tape, machine) returns accept or reject

    agenda ← \{ (Initial state of machine, beginning of tape) \}
    current-search-state ← NEXT(agenda)

    loop
        if ACCEPT-STATE?(current-search-state) returns true then
            return accept
        else
            agenda ← agenda \cup GENERATE-NEW-STATES(current-search-state)
            if agenda is empty then
                return reject
            else
                current-search-state ← NEXT(agenda)
        end

function GENERATE-NEW-STATES(current-state) returns a set of search-states

    current-node ← the node the current search-state is in
    index ← the point on the tape the current search-state is looking at

    return a list of search states from transition table as follows:
    (transition-table[current-node, ej, index])
    \cup (transition-table[current-node, tape[index+j], index + 1])

function ACCEPT-STATE?(search-state) returns true or false

    current-node ← the node search-state is in
    index ← the point on the tape search-state is looking at

    if index is at the end of the tape and current-node is an accept state of machine
    then
        return true
    else
        return false
Algorithm for Non-Deterministic FSAs

• Implementation of the NEXT function
  – **Depth-first search or Last In First Out (LIFO)**
    • Place the newly created states at the front of the agenda
    • The NEXT returns the state at the front of the agenda
  – **Breadth-first search or First In First Out (FIFO)**
    • Place the newly created states at the back of the agenda
  – **Dynamic programming or A**

Time-synchronous
Viterbi/Breadth-first search

Time-asynchronous
Best-first search
Algorithm for Non-Deterministic FSAs

- Depth-first search

![Diagram of depth-first search process with states and transitions, including agendas with specific positions and actions like b a a a! NIL.]
Algorithm for Non-Deterministic FSAs

- Breadth-first search

\[
\begin{array}{c}
1 & b & a & a & a & ! \\
2 & b & a & a & a & ! \\
3 & b & a & a & a & ! \\
4 & b & a & a & a & ! \\
5 & b & a & a & a & ! \\
\end{array}
\]

\[
\begin{array}{c}
(q_0, \text{pos 0}) \\
(q_1, \text{pos 1}) \\
(q_2, \text{pos 2}) \\
(q_3, \text{pos 3}) \\
(q_3, \text{pos 4}) \\
\end{array}
\]

\[
\begin{array}{c}
(q_0, \text{pos 0}) \\
(q_1, \text{pos 1}) \\
(q_2, \text{pos 2}) \\
(q_3, \text{pos 3}) \\
(q_3, \text{pos 4}) \\
\end{array}
\]
Relating DFSA and NDFSA

• For any NFSA, there is an exactly equivalent DFSA (which has the same power)
  – A simple algorithm for converting an NFSA to an equivalent DFSA
    • E.g. a parallel algorithm traverses the NFSA and groups the states we reach on the same input symbol into an equivalent class and give a new state label to this new equivalent class state
  – The number of states in the equivalent deterministic automaton may be much larger
Relating DFSA and NDFSA
Regular Languages and FSAs

• Regular languages
  – The class of languages that are **definable** by regular expressions
  – Or the class of languages that are **characterized** by finite-state automata

• **Definition of regular language**
  – ∅ is a primitive regular language
  – ∀a ∈ Σ ∪ ε, \{a\} is a primitive regular language
  – If L₁ and L₂ are regular languages, then so are
    • L₁ · L₂ = \{xy| x ∈ L₁, y ∈ L₂\} the **concatenation** of L₁ and L₂
    • L₁ ∪ L₂ the **union** or **disjunction** of L₁ and L₂
    • L₁* the **Kleene closure** of L₁
The Closure of Regular Languages

- **Interaction**: if $L_1$ and $L_2$ are regular languages then so is $L_1 \cap L_2$
- **Difference**: if $L_1$ and $L_2$ are regular languages then so is $L_1 - L_2$
- **Complementation**: if $L_1$ is a regular language then so is $\Sigma^* - L_1$
- **Reversal**: if $L_1$ is a regular language then so is $L_1^R$
The Concatenation of Two FSAs

- Accept a string consisting of a string from language $L_1$ followed by a string from language $L_2$
The Kleene * Closure of an FSA

- All final states of the FSA back to the initial states by $\varepsilon$-transitions
The Union of Two FSAs

- Accept a string in either of two languages
Review: English Morphology

• Morphology is the study of the ways that words are built up from smaller meaningful units called morphemes.

• Morphemes are divided into two classes:
  – Stems: The core meaning bearing units.
  – Affixes: Bits and pieces that adhere to stems to change their meanings and grammatical functions.

• Two classes of ways to form words from morphemes:
  – Inflectional morphology.
  – Derivational morphology.
Morphology Parsing

• Find the morphology structure of an input (surface) form

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Morphological Parsed Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td>cat + N + PL</td>
</tr>
<tr>
<td>cat</td>
<td>cat + N + SG</td>
</tr>
<tr>
<td>cities</td>
<td>city + N + PL</td>
</tr>
<tr>
<td>geese</td>
<td>goose + N + PL</td>
</tr>
<tr>
<td>goose</td>
<td>(goose + N + SG) or (goose + V)</td>
</tr>
<tr>
<td>gooses</td>
<td>gooses + V + 3SG</td>
</tr>
<tr>
<td>merging</td>
<td>merge + V + PRES-PART</td>
</tr>
<tr>
<td>caught</td>
<td>(catch + V + PAST-PART) or (catch + V + PAST)</td>
</tr>
</tbody>
</table>

word stems and morphological features
Constituents of Morphology Parser

• Lexicon
  – List of stems and affixes, with basic information about them
  – E.g.: noun/verb stems, etc.

• Morphotactics
  – The model of morpheme ordering
  – E.g.: the rule that English plural morpheme follows the noun rather than preceding it

• Orthographic rules
  – The **spelling rules** used to model the changes that occur in a word, when two morphemes combine
  – E.g: city + -s → cities ("consonant" + "y" → "ie")
FSAs for Morphotactics Knowledge

• An FSA for English nominal/verb inflection

  - Govern the ordering of affixes
FSAs for Morphototatics Knowledge

- An FSA for English adjective derivation
FSAs for Morphological Recognition

• Determine whether an input string of letters makes up a legitimate word

• An FSA for English nominal inflection
  – Plug in “sub-lexicons” into the FAS
  • E.g.: the reg-noun-stem, irreg-sg-noun etc.
Finite State Transducer (FST)

• FST has a more general function than FSA
  – FSA defines a formal language by defining a set of strings
  – FST defines a relation between two set of string
    • Add another tape
    • Add extra symbols (outputs) to the transitions (the Mealy machine)
    • Read one string and generate another one
      - E.g.: On one tape we read “cats”, on the other we write “cat +N +PL (morphology parsing)
Finite State Transducer

• Formal Definition
  - \( Q \): The set of states, \( Q = \{q_0, q_1, \ldots, q_N\} \)
  - \( \Sigma \): A finite alphabet of complex symbols, \( i:o \); \( i \) from an input alphabet \( I \), and \( o \) from an output alphabet \( O \), both include the epsilon symbol \( \varepsilon \)
  - \( q_0 \): the start state
  - \( F \): the set of accept/final states
  - \( \delta(q,i:o) \): the transition function that maps \( Q \times \Sigma \) to \( Q \)

• FST are closed under union, but not closed under difference, complement, and intersection (because of epsilon symbol \( \varepsilon \), et al.)
Finite State Transducer

- Two additional closure properties
  - **Inversion**
    - The inversion of a transducer $T (T^{-1})$ simply switches the input and output labels.
    - FST-as-parser $\leftrightarrow$ FST-as-generator
  - **Composition**
    - If $T_1$ is a transducer from $I_1$ to $O_1$ and $T_2$ a transducer from $I_2$ to $O_2$ then $T_1 \circ T_2$ maps from $I_1$ to $O_2$.
Two-level Morphology System

- Generating and Parsing with FST lexicon and rule

Lexical

Intermediate

Surface

LEXICON–FST

generating a string

parsing a string (more complicated)
Two-level Morphology System

• Orthographic rules
  – An FST to process a sequence of words
  • #: word boundary

Intermediate \{ f \ o \ x \ ^ \ s \ # \} 
Surface \{ f \ o \ x \ e \ s \}